# The Beauty of Software Portability

An article by Space Banana

Cross-platform software is more common than ever, now that more and more people are using different operating systems. There's plenty of systems you can port your software to, and there are lots of software projects that aim to be as portable as possible, so you can use them in your specific system. Not all developers care about cross-platform though, and there are those who care but their software relies on platform-specific implementations. Let's see then how software can be cross platform.

## Cross-platform and Portability

Cross-platform software is a software project that is distributed to multiple systems. For a piece of software to be cross-platform, it needs to be portable enough for the developers to distribute it to multiple platforms.

Software portability is how compatible a software project is to other systems, and how easy it is to make it even more compatible.

## Making your software portable

Software portability mostly comes down to external programs, libraries and APIs your project might depend on. Platform-specific implementations are the enemy of portability, as these implementations rely on a specific platform or set of platforms and cannot be used on different systems. By using platform-specific implementations, you now must decide whether you will not support other platforms, or you will implement their own equivalent implementation.

Imagine you are developing a CLI or TUI software without any external dependencies. This software is extremely portable, as it doesn't use any platform-specific implementations, or rely on libraries that use those. You can port your project to countless operating systems.
You now want your terminal output/screen to have colors, so you use ANSI escape codes. While your project is still extremely portable, you added your first platform-specific implementation: ANSI. The vast majority of systems support ANSI though, so it shouldn't be a concern of yours.

You might eventually want your software to make use of third-party software. Your portability now depends on the said software's portability.

You might want to make a GUI application instead, and so you now rely on whatever platforms your framework supports. There are a few cross-platform GUI frameworks.
Are you going to make use of graphical APIs? Make sure they are open-source and portable. OpenGL, OpenCL and Vulkan fit that category, while APIs like DirectX and CUDA are proprietary.

## Platform Freedom

Open source software can be compiled by others for their operating system and architecture. Many cases happen where the developers only build their software for 1 particular operating system and CPU architecture, but the project repository is open source and it has information on how to build from source. The end-users can therefore build the software for their own operating systems (assuming the software works on them) and CPU architectures. Source-based package managers (such as FreeBSD's freshports, NixOS's Nix and Gentoo's Portage) can build the software from source automatically.

This freedom to compile software ourselves contributes to the cross-platform nature of the project in question, and we also lose the awkward discomfort of not having enough software ported for our less popular architectures, such as ARM and RISC-V.

## Machine Code and Bytecode

There are 2 types of software compilation: compiling to machine code and compiling to bytecode.

When you compile a piece of software to machine code, the compiler reads your source code and creates a resulting binary/library file. This file is commonly known as an executable, and its data is made of CPU instructions and system calls. For this reason, binaries and libraries have 2 mandatory dependencies: a CPU architecture and an operating system kernel. If we compile a program into machine code for the Linux kernel and x86_64 CPU architecture family. The binary relies on those to run. Assuming all dependencies specific to your project are met, You can run it on any operating system that uses the Linux kernel. You have to run this software on an x86_64 machine.

Like explained in the chapter above, compiling from source lets us rely less on what the developer officially distributes, and so there are less concerns about for what system and CPU architecture the developer distributes to.

In bytecode compilation, the compiler reads your source code and creates a resulting file that is composed of instructions for a virtual machine. This software is not native to your OS and CPU, and therefore is not an executable and cannot function by itself. You run the bytecode program with the respective virtual machine it was built for. The virtual machine reads the instructions from the program and works according to them. A common example of software like this is software built for the Java Virtual Machine.
Bytecode's big advantage over machine code is the ability to be, in theory, OS-agnostic and CPU-agnostic. As long as you have the virtual machine installed in your system, your program will launch, regardless of your OS or CPU architecture.

## Reducing friction in compilation

Sometimes, developers have total power to distribute their software to countless systems, and yet they distribute only to one or two. Sometimes, you want to compile the project from source yourself, but the hassle is so big that you no longer want to do it. This tends to be caused by the friction you have when compiling software.

You have compilation friction when it's a big hassle to set up the development environment to successfully build a software project. Maybe the tooling varies depending on your system, or you need to do an obscure configuration to cross-compile, or none of the dependencies needed are included in the repository and must be installed separately.

The worst cases of friction and hassle I see usually come from C and C++ projects. C and C++ do not have an official compiler, and so the tooling varies from project to project. Is that a bad thing? Well, normally no, but this becomes a problem when the tooling varies between operating systems.
Windows C++ developers have a tendency to only set up a development environment for Windows that relies on Visual Studio for building and distributing software. Sometimes you see cross-platform software where the build instructions for Linux use Cmake, but the build instructions for Windows use Visual Studio, even though Cmake supports Windows! The tooling should be agnostic to the OS the developer or person uses, and it should not rely on IDEs and code editors! A person should only have to install the compiler and build tool, and not a whole IDE.

The other problem with many C and C++ projects is their dependencies. C and C++ have no official repository and package manager for managing libraries, and there's no third party equivalent made by the community. If the projects include the libraries' source code in the repository, everything is fine, but the problem comes when they do not. Usually, C and C++ repositories do not include their dependencies, and rely on the user to install them.
The developers make build instructions that are not system-agnostic. Using Linux as an example, a developer might make build instructions for a specific Linux system and version. The problem is if you are not in that system. In that case, the command to install all dependencies from your system's repository will be invalid, as the package names for the libraries differ between systems and versions, as well as their fragmentation! You try your best, but in the end the build fails due to missing dependencies. This is a horrible case of dependency hell.

C and C++ compilers and build tools are also notorious for their complicated processes to make cross compiling work. If developers won't bother to cross-compile, the software is not distributed to more platforms that work.

All of this hassle makes users not want to compile others' software, and makes developers not want to distribute their software to multiple systems and architectures. Our compilers or build tools should make it easier to cross-compile, so we do not have to rely on containers or virtual machines to do so.

Our project's dependencies should be included in the repository, and if they are not included then we should use a library repository, so obtaining libraries is the same for everyone, regardless of platform.

## Beware the CLI

Any functionality that you cannot build from source is dangerous for software portability. Any code you write is under your control, and libraries and frameworks written for your programming language are also partially under your control, or fully in case you master their source.

A CLI, however, is not. Many software projects, like mine, use external CLI software for specific functionality. Relying on the user to install the software raises the amount of loose dependencies your program has. You can always distribute the binary of the CLI you make use of, and execute that one, but you then have to distribute a binary for multiple operating systems and architectures. You have to study the CLI's project, build tool and be able to either cross-compile that project or prepare yourself to have multiple virtual machines.

It is also important that, if you rely on external software and interact with them through their CLI, your external dependency should be as platform-agnostic as possible. If you want to do media encoding and processing, you can use FFmpeg's CLI, and FFmpeg is extremely platform-agnostic. However, if you implement a wrapper around package managers, this wrapper is platform-specific. Are you writing a wrapper for APT? Then that will only work on Debian and Debian-based systems. Do you want to support more operating systems? Then get ready to wrap around Pacman, Nix, DNF/YUM, PKG, APK, Flatpak, etc. That's not too bad either, you could invest effort in writing a library focused on wrapping around the CLIs of package managers, that would be an interesting project, and useful for a niche of developers. That's even a project I'm considering starting.

Despite everything, you should avoid implementing certain functionality through external CLI software when said functionality is available natively in your language, in the standard library or a third-party one. Don't do file operations with CLI software such as ls, cp and dd. Use insteadd your standard library, because it provides native functionality for such things.

If you decide to make use of external software, make sure your command execution is as platform-agnostic as possible. Avoid shell commands, embrace system commands. System commands are executed as kernel system calls, and are solely composed of the command name or path to its binary and its respective CLI arguments. A shell command runs on top of a shell, and inherits its syntax and quirks, as well as additional overhead. On C an C++, make use of the exec() functions in unix-like systems and the Windows API for Windows software execution. Avoid using the built-in system() function, as that runs a shell command. Regardless, make sure you run arrays/vectors/collections as commands, rather than strings.

Imagine the following FFmpeg command:

ffmpeg -i old image.png new image.webp

If you run this as a shell command, it will not work, as the shell will intrpret "old image.png" as 2 different arguments, "old" and "image.png", and the same for "new image.webp". Running a string as a command will result in this problem, it is not whitespace-safe. Consider instead running a collection, such as an array. In the Scala language, it would look like this:

Vector("ffmpeg", "-i", "old image.png", "new image.webp")

As you can see here, each element of the vector corresponds to its own command line argument, and so the command execution is whitespace-safe, shell-independent, safe from the shell's syntax and quirks and very likely results in a kernel system call directly, but that will depend on each language's implementation.

## Conclusion

---

Software portability is an art by itself, and it should be embraced. If you do want to follow the path of platform-agnostic software, you must make sure to avoid platform-specific implementations (or implement all you need if truly necessary) and configure a build system that is able to cross-compile, or at least a mostly self-contained repository that does not need to rely on system packages as build dependencies. Make sure to include your dependencies in the repository, or access some form of standardized package repository instead of making the user rely on the system's.

Beware platform-specific and shell-specific implementations when runnning external CLI software to perform the functionality you need, and make sure all command executions are whitespace-safe.

If you depend on frameworks writtten by others, always check if they achieve the portability you require.

Even with all the care in the world, sometimes we need to implement platform-specific functionality, especially at low level programming, or our entire project never was meant to run on all platforms, but only satisfy your requirements for a specific platform.